DTIC FILE COPY

(4)

Special Report 89-8

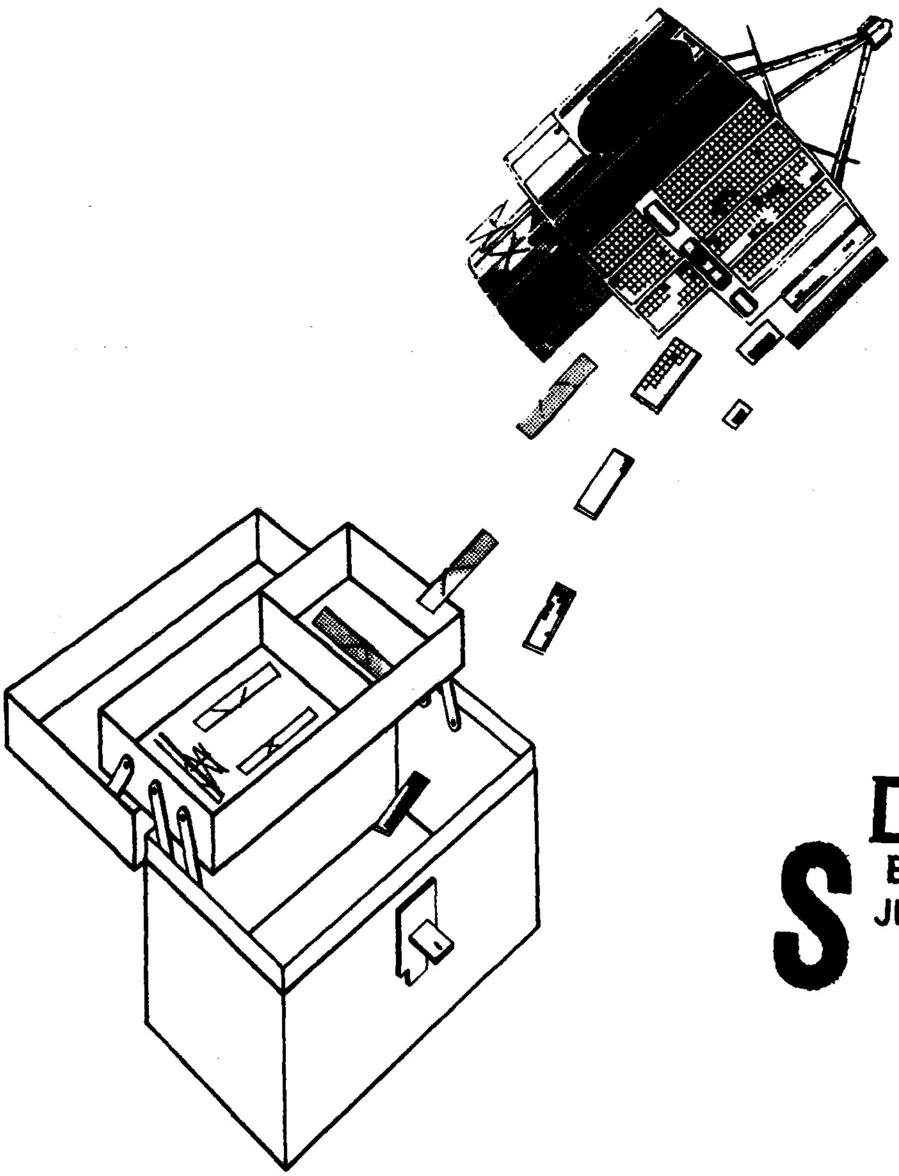
April 1989

AD-A208 945

US Army Corps of Engineers
Cold Regions Research & Engineering Laboratory

QuickDraw data structures for image processing

Perry J. LaPotin and Harlan L. McKim



Prepared for
OFFICE OF THE CHIEF OF ENGINEERS

Approved for public release; distribution is unlimited.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S) Special Report 89-8	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION U.S. Army Cold Regions Research and Engineering Laboratory 6b. OFFICE SYMBOL (if applicable) CECRL	7a. NAME OF MONITORING ORGANIZATION Office of the Chief of Engineers
6c. ADDRESS (City, State, and ZIP Code) 72 Lyme Road Hanover, N. H. 03755-1290	7b. ADDRESS (City, State, and ZIP Code) Washington, D.C. 20314
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
6.27.30A	4A762730 AT42 CWIS 32297	CS	022

11. TITLE (Include Security Classification)

QuickDraw Data Structures for Image Processing

12. PERSONAL AUTHOR(S)

LaPotin, Perry J. and McKim, Harlan L.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
	FROM ______ TO ______	April 1989	20

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

Data structures
Geographical information systems
Image processing
Parallel data structures
Parallel distributed processing
QuickDraw data structures

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Standard binary data formats are currently used to import and export satellite images to geographic information- and image-processing systems. These data structures provide a standard sequential method to read and write large volumes of information in a semicompressed format. While the binary structure is adequate for strict import and export of image data, it is poorly adapted to fast image-processing at the microcomputer level. In this study, new data structures are investigated that use operating codes to quickly convert raster binary image data and vector overlay files into a high-speed graphical language for efficient display and processing. Binary data is converted into "picture handles" of variable size and resolution using 2-byte operating codes to symbolize the graphical process. As a result, images are drawn as objects that may be coupled as independent vector components in multiple bit planes. The bit planes may be specified for each pixel to support 24- and 32-bit color of both raster and vector data. The efficiency of these structures allows the user to display 1024 x 1024 scenes in multiple overlapping windows using simple Cut/Copy/Paste commands. In addition, the use of operating codes allows an analyst to quickly store and retrieve archived images in their compressed form using simple "scrap manager" techniques. Early results for this technique indicate that converted vector overlays may be compressed by a factor of 8 and SPOT images (depending on scene diversity) by a factor of 2. More significantly, images that typically require 4 min to load from binary may be displayed in fractions of a second using the new display

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Harlan McKim	22b. TELEPHONE (Include Area Code) 603-646-4100 22c. OFFICE SYMBOL CECRL-RE

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

19. Abstract (cont'd). method and resultant operating codes. In its present form, the software provides a gateway for users of image data to display multiple bands of information quickly, and to vary hue, saturation, brightness, and resolution levels on the microcomputer. New utilities will include image export into the PNTG, PHCA, EPS, and TIFF formats for export compatibility with Postscript page-layout software and video image-processing systems.

PREFACE

This report was prepared by Dr. Perry J. LaPotin, of the Department of Physics and Astronomy, Dartmouth College, Hanover, New Hampshire, and Dr. Harlan L. McKim, Project Manager, USACE Civil Works Remote Sensing Program. Funding for this work was provided under DA Project 4A762730AT42, *Design, Construction, and Operations Technology for Cold Regions*, Task CS (Combat Support), Work Unit 022, *Winter Battlefield Terrain Sensors*; and under Civil Works Project CWIS 32297, *Demonstration of Satellite Digital Data in Corps Planning, Engineering and Operations Activities.*

The authors thank Timothy Pangburn, and Richard Haugen for their excellent review of the manuscript, and Dr. Michael Masuch of the Department of Computer Science, University of Amsterdam, and Jos C. Glorie of the Department of Physics and Astronomy, Dartmouth College, and the Department of Computer Science, University of Amsterdam, for their technical evaluation and suggestions concerning the data structures. The authors extend special thanks to Nancy LaPotin for her excellent review of the remote sensing material.

CONTENTS

Page

Abstract i
Preface ii
Introduction 1
Method 1
Picture files 4
Results 7
Conclusions 7
Literature cited 8
Appendix A: QuickDraw operating codes for converting binary image data into PICT and PICT2 data files 9

ILLUSTRATIONS

Figure

1. RGB color GrafPorts under QuickDraw 2
2. RGB color format under QuickDraw 3
3. The picture handle data structure used to store and display images in both raster and vector format 4
4. Memory must be allocated for the picture handle to receive the image and provide the required space 4
5. PICT and PICT2 file format for import and export of picture data 4
6. A short procedure to read a specified amount of information into a data pointer from a previously opened file 5
7. A procedure to spool picture data into a picture handle 5
8. A short procedure to write a specified amount of information from a data pointer into a previously opened file 6
9. A procedure to write PICT2-style files from a picture handle and force the required End of Picture marker 6

QuickDraw Data Structures for Image Processing

PERRY J. LAPOTIN AND HARLAN L. MCKIM

INTRODUCTION

Import and export formats for both Landsat and SPOT imagery are widely supported within the image-processing/remote-sensing community (Holkenbrink 1978, *SPOT Image* 1983). These formats provide a standard format for the flow of information between various hosts, but are poorly suited to fast image-processing at the microcomputer level (Cohen and Grossberg 1983, Winston 1984, Rumelhart and Zipser 1985, McClelland and Rumelhart 1987).

In this study, a new graphically dependent method for converting binary information into QuickDraw[1] operating codes is investigated. The technique converts pixels of variable gray scale (usually 0–255) into scaled RGB intensities. The scaled intensities are stored within a pixel map that contains information on the base address of where the information may be retrieved from memory as well as information on the size, horizontal and vertical resolution, and planar offsets (for greater than 8-bit color). In the developed prototype, pictures are referred to by their handle in memory (pointers to master pointers that point to the picture data structure in memory). These handles may appear clumsy at first, but they are needed to quickly pull large volumes of information from memory in the image display process. Furthermore, it will become apparent that they are needed for the design of efficient and dynamic data structures to display, overlay, and analyze large scenes in multiple windows (GrafPorts). GrafPorts are the "logical paper" required to display images in windows, dialogs, and on most output devices. The data structure is defined in Figure 1 and a detailed description may be found in *Inside Macintosh* (1985). GrafPorts that contain pixel maps may be quickly converted to their vector equivalent using standard picture data structures and "off-the-shelf" bit transfer routines.

METHOD

All image data is sequential, so they may be handled in a systematic manner depending on the specific input format (e.g. BSQ, BIL, BIP, BIPP). Binary data are converted to a picture handle by first "spooling-in" the information as a raster image to either an on-screen (visible to the user) or off-screen (invisible to the user) pixel map. This is accomplished in the following manner.

1. A pointer of dynamic size (or fixed to a segment size of the image) is created in memory to store the image bytes temporarily prior to display. This image size dictates the number of segments needed to read in the information. Since large scenes require large memory allocation (e.g. 1024 x 1024 requires 1,048,576 bytes), segments of variable size (e.g. 32,000 bytes) are used and the memory is freed after each read-in sequence. This implies that segment pointers are either disposed of or re-indexed (to the beginning using ordinal operators) following each segment read.

2. After reading in a segment, the binary (represented in character form) for each pixel in the segment is converted to its ordinal value in the 0–255 range. The ordinal values are used to create a color range for display of the information into the GrafPort.

3. Each pixel in the 0–255 range is converted to the RGB color range specified by QuickDraw. Therefore, each pixel in the 0–255 range requires conversion to the RGB color format of Figure 2. Note that RGB color is composed of separate red, green, and blue intensities, and a color specifica-

---

[1] QuickDraw is the graphical operating language for the Apple Macintosh. This paper assumes that the reader is familiar with structured programming techniques and has some familiarity with memory management methods. For additional information on handles and QuickDraw refer to *Inside Macintosh* (1985).

```
CDialogPtr = CWindowPtr;
CWindowPtr = CGrafPtr;

CGrafPtr = ^CGrafPort;
CGrafPort = RECORD
                 :
                 portPixMap: pixMapHandle;
                 :
                 :
            END;

pixMapHandle = ^pixMapPtr; { handle to a pixel map}
pixMapPtr = ^pixMap;
pixMap   =  RECORD
                 baseAddr : Ptr;              { pointer to pixels}
                 rowBytes : INTEGER;          { offset to next line}
                 Bounds : Rect;               { encloses bitmap}
                 pmVersion : INTEGER;         { pixMap version number}
                 packType : INTEGER;          { defines packing format}
                 packSize : LONGINT;          { length of pixel data}
                 hRes : Fixed;                { horiz. resolution (ppi)}
                 vRes : Fixed;                { vert. resolution (ppi)}
                 pixelType : INTEGER;         { defines pixel type}
                 pixelSize : INTEGER;         { no. of bits in pixel}
                 cmpCount : INTEGER;          { no. of components in pixel}
                 cmpSize : INTEGER;           { no. of bits per component}
                 planeBytes : LONGINT;        { offset to next plane}
                 pmTable : CTabHandle;        { color table for this pixMap}
                 pmReserved : LONGINT;        { for future use. MUST BE 0}
            END;
```

Figure 1. RGB color GrafPorts under QuickDraw. Pixel maps to store information on memory location (baseAddr)*, number of bytes per row* (rowBytes)*, and the minimum rectangle that bounds the image* (Bounds)*. The remaining fields are used to specify packing formats, resolution, bit planes, and indexes to color lookup tables (CLUT's).*

tion table can be set up to determine equivalent RGB color for a particular index value. Since RGB color is composed of three separate fields, a single band of the image data can be "loaded" into a single gun (e.g. only the red display) and viewed in one color range. A false representation (of the true color composite) can be created for a single band by loading the remaining two color fields with index values derived from the single 0–255 intensity. Conversely, three bands of information can be read to create a "true" false color composite. In either case, the 0–255 range needs to be converted to the integer scale (–32676 to 32676) to achieve a full dynamic range of color. For density slicing, a subrange of the 0–255 range is chosen and expanded to the integer scale. Hue, saturation, and brightness may be changed independent of the RGB color selection.

4. Pixels in the segment (e.g. the block of 32,000 bytes) are converted to their QuickDraw RGB equivalents and are displayed within an on-screen or off-screen GrafPort. A GrafPort is the basic data structure for storing, manipulating, and displaying information within a window that may contain both horizontal and vertical scroll-bars. In the QuickDraw environment, both dialogs and windows are pointers to GrafPorts and are simply special versions of the basic data structure. Figure 1 shows the *pixMap* data structure of the GrafPort using the Color QuickDraw

```
RGBColor = RECORD
            red : INTEGER;      { magnitude of red component}
            green : INTEGER;    { magnitude of green component}
            blue : INTEGER;     { magnitude of blue component}
        END;

ColorSpec = RECORD
            value : INTEGER;    { index or other value}
            rgb : RGBColor;     { true color}
        END;
```

Figure 2. RGB color format under QuickDraw. Binary data are scaled to the range of integers for separate red, green, and blue display.

environment. For the sake of clarity, only the *portPixMap* data structure is shown in its entirety. There are many fields within the color GrafPort, ranging from device specifications to fore- and background color specifications. For additional information, see *Inside Macintosh* (1987).

As previously mentioned, color windows (*CWindowPtr*) and color dialogs (*CDialogPtr*) are simply equivalent to color GrafPorts (*CGrafPtr*). The single field shown within the color GrafPort structure is the port pixel map, which contains all the specifications for the raster form of the image data. To refer to this information, a handle (the pointer to a pointer, a method called "double inflection") is used. The handle points to a "master pointer" that keeps track of memory locations.[2] In this manner, windows and dialogs are attached to GrafPorts that hold the information needed to recreate the image simply by pointing to necessary fields within the data structure. A procedure that just transfers pixel bytes may be used to swap the image in and out of memory quickly without reloading the scene or recalculating the RGB equivalent intensities.

5. Once all segments for the image are converted to their RGB equivalent in step 4 and the pixel map is loaded with the image, it either remains within the *CGrafPtr* data structure or it may be "spooled" into a picture to convert it to a vector equivalent. The vector equivalent for the rasterized pixel map is simply a vector representation of the raster block. This implies that conversion is just a conversion of data types and is limited in the actual vectorization. More in-depth vectorization, such as bezier curve tracing, can always be done after the initial raster-to-vector conversion using off-the-shelf software, such as Digital Darkroom (1988) and Adobe88 (1988).

The picture is a dynamic record of QuickDraw operating codes (opcodes)[3] listed in Appendix A. The data structure is dynamic from two perspectives:

- Its size is bounded only by available memory
- The structure is a handle and is thus a dynamic memory address.

Specifically, the data structure is defined in Figure 3. The picture definition data consider all

---

[2] One might consider the master pointer as a postman trying to deliver your mail. The postman points to your house and its contents, and the post office points to the postman to give him the mail:

```
PostOffice : ^Postman {a postoffice is a "^" pointer
        to the postman}
PostMan : ^ MyHouse {the postman is a pointer to
my house and its contents}
MyHouse = RECORD {finally, my house contains
my items and specifications}
            myCouch : SleeperType;
            myDog : HoundType;
            myCar : VWType;
        END;
```

The use of double inflection becomes important in an environment where one click of the mouse switches between windows (GrafPorts), and scrollbars pan information up and down and side to side instantaneously. Efficient data structures, efficient memory address management, and master pointing are required (Forsyth and Rada 1986).

[3] Variable-sized "action items" that trigger a drawing operation. The codes are passed directly to an output device (screen, printer) for high-speed display, and are stored in compressed form within the PICT and PICT2 format.

```
picHandle = ^picPtr;
picPtr = ^Picture;
Picture = RECORD
          picSize : INTEGER;
          picFrame : Rect;
            {then picture definition data—the
            QuickDraw opcodes}
          END;
```

Figure 3. The picture handle data structure used to store and display images in both raster and vector format.

```
SPOTPict: = picHandle(NewHandle(Sizeof(Picture))); with
SPOTPict ^^ DO
      BEGIN
      picSize := Sizeof(Picture);          {dynamic, so don't worry}
      picFrame := SPOTRect;                {e.g. 1024 x 1024}
```

Figure 4. Memory must be allocated for the picture handle to receive the image and provide the required space. The Sizeof *function creates a new handle in memory and specifies a temporary picture size. The real size of the picture expands to fit the actual size of the image.*

data to be in vector format and handle raster images as a special vector style. Therefore, images that were created as a series of arcs, lines, polygons, and other vector drawings get stored as vector simply because they were created that way. Raster images (e.g. MacPaint) are stored as raster using vector operations (e.g. a pixel is simply a rectangle of unit size).

The data structure allows for the import of imagery simply by specifying the image size (*picSize*) and the minimum bounding rectangle that encompasses the image (*picFrame*). Since the size of the structure is dynamic, *picSize* can be dynamically allocated to force-fit the image using the *Sizeof* function (standard in Pascal and C). For example, if *SPOTPict* is of type *picHandle* in Figure 3, then a dynamic picture size may be allocated using the expression *SPOTPict^^. picSize* : = *Sizeof (Picture)*. In practice, however, memory must be allocated to all assignments for the picture handle using the *NewHandle* operator (Fig. 4).

In sum, picture handles are used to swap imagery quickly in and out of memory for fast image display. Since each picture handle contains the opcodes that were used to create it, no information is lost from the original binary data. The PICT2 format for import and export of imagery is simply an extension of the picture handle data structure with a 512-byte header. This association between picture handles and the QuickDraw import/export standard provides a quick and efficient method for spooling large volumes of information into (and out of) a PICT2 data file.

PICTURE FILES

Picture handles may be quickly converted to picture documents (alias MacDraw PICT) by writing the information from the handle into an open file. Figure 5 shows the basic construction of a PICT (black and white) and a PICT2 data file (extended size and color).

Segments of a data file that contain specific information styles are called forks. Resource forks typically contain the "raw" numerical information necessary for the graphical procedure (e.g. the bits necessary to draw an icon or the specification necessary to show a dialog box). For the PICT2 format, only the data fork contains any information and the resource fork is empty. Within the data fork, the format of the file is sequential of the form: 1) a 512-byte header, 2) the picture size in bytes, 3) a minimum bounding rectangle, 4) the opcodes, and 5) an End of Picture Marker = $00FF.

To import imagery from an open picture file,

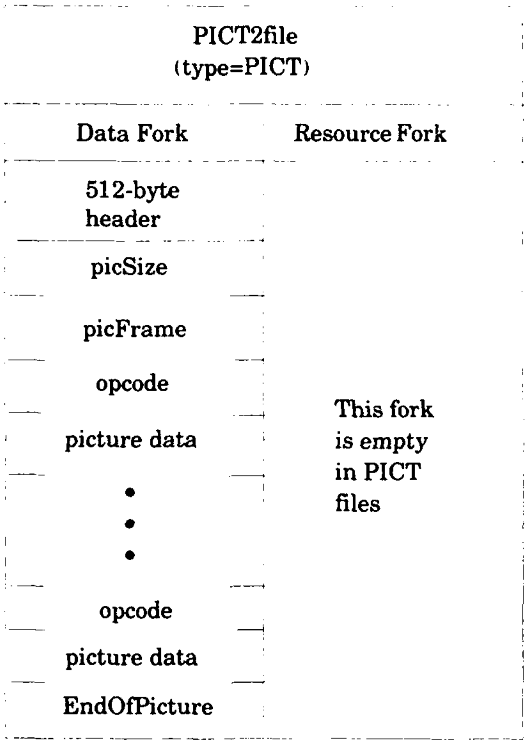


Figure 5. PICT and PICT2 file format for import and export of picture data.

a spooling routine is used to handle the header and read the opcodes into a previously allocated picture handle. An outline of this method is provided in Figures 6 and 7. Figure 6 is a short procedure that simply reads in information from a previously open file whose reference number is the variable *Refnum*. In Figure 7, this procedure is used as a graphical procedure to spool in the information from the file.

Images are exported from a picture handle in the reverse manner from how they were read into the picture handle. Figures 8 and 9 give examples of this process using the procedure P*utPICTData* as the graphical procedure for the write process, and the function *WriteImage* to write the image from the picture handle (*thePict*) into a PICT2 file whose reference number is *RefNum*.

In sum, the PICT2 data file is a simple extension of the picture handle data structure provided in Figure 3. It represents a powerful data format for processing large volumes of information from files that are flexible both in size and in data structure (raster and vector). In addition, the format is recognized as a standard for the import and export of graphical information to other image-processing software.

```
PROCEDURE GetPictData (dataPtr : Ptr; byteCount : longint);
VAR
        longCount : LONGINT;
        ReadErr : OSErr;
BEGIN

        longCount := byteCount;
        ReadErr := FSRead(RefNum, longCount, dataPtr);
        {no readErr handling here...}
END;{GetPictData}
```

Figure 6. A short procedure to read a specified amount of information (byteCount) *into a data pointer* (dataPtr) *from a previously opened file* (RefNum).

```
FUNCTION ReadPICT; {(RefNum : INTEGER; aWindow :  WindowPtr) : boolean;{ }
CONST
                PICThead = 512; {512 bytes to start things off}
VAR
                ReadErr : OSErr;
                Count: LONGINT;
                thePict : picHandle;
                myReadProcs : cqdProcs;
BEGIN
        MaxApplZone;
        ReadErr := SetFPos(RefNum, fsFromStart, 0);{ }
        SetStdCProcs(myReadProcs);{ }
        aWindow^.grafProcs := @myReadProcs;{ }
        myReadProcs.putPicProc := @GetPICTData;{ }
                {for now, skip the header and read the size of the picture field; we can discard this since
                the bounds of the picture frame are actually used to determine the number of bytes to
                be read into the picture handle...}
        ReadErr : = SetFPos(RefNum, fsFromStart, PICThead);
                {create room in memory for the new picture}
        thePict := picHandle(NewHandle(Sizeof(Picture)));{ }
        Count := Sizeof(Picture);

        IF (FSRead(RefNum, Count, Ptr(thePict^)) = noErr) THEN{ }
                BEGIN
                        ReadErr := SetFPos(RefNum, fsFromStart, PICThead);
                        Count := thePict^^.picSize;

                        IF (FSRead(RefNum, Count, Ptr(thePict^)) = noErr)
                THEN ReadPICT := true
                        ELSE {handle the bad read and the bad news...}
                        ReadPICT := false;
                END;

        aWindow^.grafProcs := NIL;{ }
        ReadErr := FSClose(RefNum);
END;{ReadPICT}
```

Figure 7. A procedure to spool picture data into a picture handle (thePict). *The procedure uses* GetPICTData *(Fig. 6) as a graphical procedure and* FSRead (Inside Macintosh *1987) to read in the basic data (operating codes).*

```
PROCEDURE PutPICTData (dataPtr : Ptr; byteCount : integer);
VAR
          WriteErr : OSErr;
          longCount : LONGINT;
BEGIN
  longCount := byteCount;
  WriteErr := FSWrite(RefNum, longCount, dataPtr);{ }
          {handle errors in WriteImage}
END;{PutPICTData}
```

Figure 8. A short procedure to write a specified amount of information (byteCount) *from a data pointer* (dataPtr) *into a previously opened file* (RefNum).

```
FUNCTION WriteImage (RefNum : integer) : OSErr;
CONST
            PICThead = 512; {512 bytes to start things off}
            PICTtail = 2;
TYPE
            DiskBlk = PACKED ARRAY[1..BlokSize] OF qdByte;
VAR
            WriteErr : OSErr;
            dstBuf : DiskBlk;
            Count : LONGINT;
            thePict : PicHandle;
            myProcs : cqdProcs;
            TailBlk : ARRAY[1..PICTtail] OF Byte;
BEGIN
            WriteErr := SetFPos(RefNum, fsFromStart, 0);
            SetStdCProcs(myProcs);
            aWindow^.grafProcs := @myProcs;
            myProcs.putPicProc := @PutPICTData;
            Count := PICThead;
                  {place all translation information in the dstBuf so that
                  it can be written into the picture header. This tells the
                  software how the image was converted from binary to
                  PICT for future reference, and rewrite back to binary}
            WriteErr := FSWrite(RefNum, Count, @dstBuf);
                  {thePict is the picture handle with opcodes}
            Count := Sizeof(thePict);
            WriteErr := FSWrite(RefNum, Count, Ptr(thePict^));{ }
                  {write EOF opcode for PICT2 files}
            TailBlk[1] := $00;
            TailBlk[2] := $FF;

            Count := PICTtail;
            WriteErr := FSWrite(RefNum, Count, @TailBlk);

            aWindow^.grafProcs := NIL;
                  {handle write errors elsewhere}
            WriteImage := WriteErr;
END;{WriteImage}
```

Figure 9. A procedure to write PICT2-style files from a picture handle (thePict) *and force the required End Of Picture marker. The 512-byte header documents how the image was created from the original binary (e.g. the conversion from the original 0–255 used to create the RGB display). If additional information is required to document the picture, the resource fork (Fig. 5) is used as the depository for the information.*

RESULTS

A prototype application has been developed to convert image data into PICT2 format. While this application is still in its infancy, results indicate that there is a significant improvement in microcomputer image-processing performance as a result of the conversion. Early benchmarks for a 512 x 512 SPOT image on a Macintosh II (with 2 megabytes of memory and 3 bands open) suggest that images may be displayed and redrawn in fractions of a second. Comparable images typically require 4 minutes to load from binary and are ill-suited for dynamic display in multiple windows. Early results for data compression indicate that vector overlays may be compressed in memory by a factor of 8 (from comparable MOSS polygon files) and SPOT images (depending on scene diversity) by a factor of 2. Data compression for import and export indicates that images may be saved in the PICT2 format at roughly 90% of their binary size (again depending on image diversity). Preliminary testing of the software illustrates that images may be displayed in multiple overlapping windows with active vertical and horizontal scrolling of RGB color. Furthermore, images may be cut, copied, and pasted at variable zoom and resolution levels between windows and the scrapbook using standard "scrap management" and QuickDraw procedures.

CONCLUSIONS

While the PICT2 format will not replace the binary structure for the strict import and export of image data, it accelerates greatly the speed and memory performance of digital image processing on certain hosts. In this study, data structures have been derived that may be used to process images and import or export image data in a format compatible with other commercial software. Advantages of this approach include:

1) compatibility with most PostScript- and QuickDraw-based software,

2) very fast graphical processing,

3) flexible data compression,

4) flexible data structuring (dynamic size and memory allocation),

5) no loss of information from original binary form, and

6) flexible resolution, hue, saturation, brightness, and bit-plane specifications.

Future efforts will include faster data conversion into the PICT2 format by generating operating codes directly from the input stream (they are currently generated from the GrafPort that displays the input stream). This should accelerate binary-to-picture conversion by a factor of 2 and eliminate the need to use the GrafPort as an intermediate buffer. In addition, new utilities will include image export into the PNTG, PHCA, EPS, and TIFF formats. These formats provide an export technique for quick image-processing using video (TIFF) and page-layout software (PNTG, PHCA). Furthermore, the Postscript compatibility (EPS) will provide a convenient route for exporting images into UNIX and other operating systems that support Postscript but not QuickDraw.

LITERATURE CITED

Adobe88 (1988) Adobe Illustrator, Adobe Systems Inc.

Cohen, M. A. and S. Grossberg (1983) "Absolute stability of global pattern formation and parallel memory storage by competitive neural networks," in IEEE Transactions on Systems, Man and Cybernetics, **SMC-13**(5):815-824.

Digital Darkroom (1988) Silicon Beach Software.

Forsyth, R. and R. Rada (1986) *Machine Learning: applications in expert systems and information retrieval.* Chichester, England: Ellis Horwood Limited.

Holkenbrink, P. F. (1978) "Manual on Characteristics of Landsat Computer-Compatible Tapes Produced by the EROS Data Center Digital Image Processing System," Version 1.0, USGS 024-001-03116-7, Washington, D.C.

Inside Macintosh (1985) Promotional Edition, Apple Computer, Cupertino, Calif.

Inside Macintosh (1987) Volume V, Final APDA Draft, Apple Programmer's and Developer's Association, APDA KMB15F, Apple Computer, Cupertino, Calif.

McClelland, J. L. and D. E. Rumelhart (1987) *Parallel Distributed Processing*, Vol 1: Foundations. Cambridge, Mass.: MIT Press.

Rumelhart, D. E. and D. Zipser (1985) "Feature discovery by competitive learning," in *Cognitive Science*, **9**(75-112).

SPOT Image (1983) 1983 U.S. SPOT Simulation Campaign Auxiliary Information Package, Washington, D.C.: SPOTIMAGE Corp.

Winston, P. H. (1984) *Artificial Intelligence*, 2nd Ed., Reading, Mass.: Addison Wesley.

APPENDIX A

QuickDraw Operating Codes (opcodes) for converting binary image data into PICT and PICT2 Data Files.

Source: *Inside Macintosh* (1987)

Table A. Data types.

Type	*Size*
v1 opcode	1 byte
v2 opcode	2 bytes
integer	2 bytes
long integer	4 bytes
mode	2 bytes
point	4 bytes
0...255	1 byte
–128...127	1 byte (signed)
rect	8 bytes (top, left, bottom, right: integer)
poly	10+ bytes
region	10+ bytes
fixed-point number	4 bytes
pattern	8 bytes
rowBytes	2 bytes (always an even quantity)

Table B. PICT opcodes.

Opcode[1,2]	*Name*	*Description*	*Data Size (in bytes)*
$0000	NOP	nop	0
$0001	Clip	clip	region size
$0002	BkPat	background pattern	8
$0003	TxFont	text font (word)	2
$0004	TxFace	text face (byte)	1
$0005	TxMode	text mode (word)	2
$0006	SpExtra	space extra (fixed point)	4
$0007	PnSize	pen size (point)	4
$0008	PnMode	pen mode (word)	2
$0009	PnPat	pen pattern	8
$000A	FillPat	fill pattern	8
$000B	OvSize	oval size (point)	4
$000C	Origin	dh, dv (word)	4
$000D	TxSize	text size (word)	2
$000E	FgColor	foreground color (long)	4
$000F	BkColor	background color (long)	4
$0010	TxRatio	numer (point), denom (point)	8
$0011	Version	version (byte)	1
$0012	*BkPixPat	color background pattern	variable: see Table C
$0013	*PnPixPat	color pen pattern	variable: see Table C
$0014	*FillPixPat	color fill pattern	variable: see Table C
$0015	*PnLocHFrac	fractional pen position	2
$0016	*ChExtra	extra for each character	2
$0017	*reserved for Apple use opcode[3,4]		0
$0018	*reserved for Apple use opcode		0
$0019	*reserved for Apple use opcode		0
$001A	*RGBFgCol	RGB foreColor	variable: see Table C
$001B	*RGBBkCol	RGB backColor	variable: see Table C

Table B (cont'd).

Opcode	Name	Description	*Data Size (in bytes)*
$ 001C	*HiliteMode	hilite mode flag	0
$001D	*HiliteColor	RGB hilite color	variable: see Table C
$001E	*DefHilite	Use default hilite color	0
$001F	*OpColor	RGB Opcolor for arithmetic modes	variable: see Table C
$0020	Line	pnLoc (point), newPt (point)	8
$0021	LineFrom	newPt (point)	4
$0022	ShortLine	pnLoc (point, dh, dv (–128...127)	6
$0023	ShortLineFrom	dh, dv (–128...127)	2
$0024	*reserved for Apple use opcode[3,4] + 2 bytes data length + data		2+ data length
$0025	*reserved for Apple use opcode + 2 bytes data length + data		2+ data length
$0026	*reserved for Apple use opcode + 2 bytes data length + data		2+ data length
$0027	*reserved for Apple use opcode + 2 bytes data length + data		2+ data length
$0028	LongText	txLoc (point), count (0..255), text	5 + text
$0029	DHText	dh (0...255), count (0..255), text	2 + text
$002A	DVText	dv (0...255), count (0...255), text	2 + text
$002B	DHDVText	dh, dv (0...255), count (0.255), count, text	3 + text
$002C	*reserved for Apple use opcode[3,4] + 2 bytes data length + data		2+ data length
$002D	*reserved for Apple use opcode + 2 bytes data length + data		2+ data length
$002E	*reserved for Apple use opcode + 2 bytes data length + data		2+ data length
$002F	*reserved for Apple use opcode + 2 bytes data length + data		2+ data length
$0030	frameRect	rect	8
$0031	paintRect	rect	8
$0032	eraseRect	rect	8
$0033	invertRect	rect	8
$0034	fillRect	rect	8
$0035	*reserved for Apple use opcode[3,4] + 8 bytes data		8
$0036	*reserved for Apple use opcode + 8 bytes data		8
$0037	*reserved for Apple use opcode + 8 bytes data		8
$0038	frameSameRect	rect	0
$0039	paint SameRect	rect	0
$003A	eraseSameRect	rect	0
$003B	invertSameRect	rect	0
$003C	fillSameRect	rect	0
$003D	*reserved for Apple use opcode[3,4]		0
$003E	*reserved for Apple use opcode		0
$003F	*reserved for Apple use opcode		0
$0040	frameRRect	rect[5]	8
$0041	paintRRect	rect[5]	8
$0042	erase RRect	rect[5]	8
$0043	invertRRect	rect[5]	8
$0044	fillRRect	rect[5]	8
$0045	*reserved for Apple use opcode[3,4] + 8 bytes data		8
$0046	*reserved for Apple use opcode + 8 bytes data		8
$0047	*reserved for Apple use opcode + 8 bytes data		8

Table B (cont'd). PICT opcodes.

Opcode	*Name*	*Description*	*Data Size (in bytes)*
$0048	frameSameRRect	rect	0
$0049	paintSameRRect	rect	0
$004A	eraseSameRRect	rect	0
$004B	invertSameRRect	rect	0
$004C	fillSameRRect	rect	0
$004D	*reserved for Apple use opcode[3,4]		0
$004E	*reserved for Apple use opcode		0
$004F	*reserved for Apple use opcode		0
$0050	frameOval	rect	8
$0051	paintOval	rect	8
$0052	eraseOval	rect	8
$0053	invertOval	rect	8
$0054	fillOval	rect	8
$0055	*reserved for Apple use opcode[3,4] + 8 bytes data		8
$0056	*reserved for Apple use opcode + 8 bytes data		8
$0057	*reserved for Apple use opcode + 8 bytes data		8
$0058	frameSameOval	rect	0
$0059	paintSameOval	rect	0
$005A	eraseSameOval	rect	0
$005B	invertSameOval	rect	0
$005C	fillSameOval	rect	0
$005E	*reserved for Apple use opcode[3,4]		0
$005F	*reserved for Apple use opcode		0
$0060	frameArc	rect, startAngle, arcAngle	12
$0061	paintArc	rect, startAngle, arcAngle	12
$0062	eraseArc	rect, startAngle, arcAngle	12
$0063	invertArc	rect, startAngle, arcAngle	12
$0064	fillArc	rect, startAngle, arcAngle	12
$0065	*reserved for Apple use opcode [3,4]+ 12 bytes data		12
$0066	*reserved for Apple use opcode + 12 bytes data		12
$0067	*reserved for Apple use opcode + 12 bytes data		12
$0068	frameSameArc	rect	4
$0069	paintSameArc	rect	4
$006B	invertSameArc	rect	4
$006C	fillSameArc	rect	4
$006D	*reserved for Apple use opcode[3,4] + 4 bytes data		4
$006E	*reserved for Apple use opcode + 4 bytes data		4
$006F	*reserved for Apple use opcode + 4 bytes data		4
$0070	framePoly	poly	polygon size
$0071	paintPoly	poly	polygon size
$0072	erasePoly	poly	polygon size
$0073	invertPoly	poly	polygon size
$0074	fillPoly	poly	polygon size
$0075	*reserved for Apple use opcode[3,4] + poly		
$0076	*reserved for Apple use opcode + poly		
$0077	*reserved for Apple use opcode word + poly		
$0078	frameSamePoly	(not yet implemented: same as 70, etc.)	0
$0079	paintSamePoly	(not yet implemented)	0
$007A	eraseSamePoly	(not yet implemented)	0
$007B	invertSamePoly	(not yet implemented)	0
$007C	fillSamePoly	(not yet implemented)	0
$007D	*reserved for Apple use opcode[3,4]		0
$007E	*reserved for Apple use opcode		0
$007F	*reserved for Apple use opcode		0

Table B (cont'd).

Opcode	*Name*	*Description*	*Data Size (in bytes)*
$0080	frameRgn	rgn	region size
$0081	paintRgn	rgn	region size
$0082	eraseRgn	rgn	region size
$0083	invertRgn	rgn	region size
$0084	fillRgn	rgn	region size
$0085	*reserved for Apple use opcode[3,4] + rgn		region size
$0086	*reserved for Apple use opcode + rgn		region size
$0087	*reserved for Apple use opcode + rgn		region size
$0088	frameSameRgn	(not yet implemented - same as 80, *etc.*)	0
$0089	paintSameRgn	(not yet implemented)	0
$008A	eraseSameRgn	(not yet implemented)	0
$008B	invertSameRgn	(not yet implemented)	0
$008C	fillSameRgn	(not yet implemented)	0
$008D	*reserved for Apple use opcode[3,4]		0
$008E	*reserved for Apple use opcode		0
$008F	*reserved for Apple use opcode		0
$0090	*BitsRect	copybits, rect clipped[6]	variable[8]
$0091	*BitsRgn	copybits, rgn clipped[6]	variable[8]
$0092	*reserved for Apple use opcode [3,4]+ 2 bytes data length + data		2+ data length
$0093	*reserved for Apple use opcode + 2 bytes data length + data		2+ data length
$0094	*reserved for Apple use opcode + 2 bytes data length + data		2+ data length
$0095	*reserved for Apple use opcode + 2 bytes data length + data		2+ data length
$0096	*reserved for Apple use opcode + 2 bytes data length + data		2+ data length
$0097	*reserved for Apple use opcode word + 2 bytes data length + data		2+ data length
$0098	*PackBitsRect	packed copybits, rect clipped	variable[8]
$0099	*PackBitsRgn	packed copybits, rgn clipped	variable[8]
$009A	*reserved for Apple use opcode [3,4]+ 2 bytes data length + data		2+ data length
$009B	*reserved for Apple use opcode + 2 bytes data length + data		2+ data length
$009C	*reserved for Apple use opcode + 2 bytes data length + data		2+ data length
$009D	*reserved for Apple use opcode + 2 bytes data length + data		2+ data length
$009E	*reserved for Apple use opcode + 2 bytes data length + data		2+ data length
$009F	*reserved for Apple use opcode + 2 bytes data length + data		2+ data length
$00A0	ShortComment	kind (word)	2
$00A1	LongComment	kind (word), size (word), data	4+ data
$00A2	*reserved for Apple use opcode [3,4]+ 2 bytes data length + data		2+ data length
⋮		⋮	
$00AF	*reserved for Apple use opcode + 2 bytes data length + data		2+ data length

Table B (cont'd). PICT opcodes.

Opcode	*Name*	*Description*	*Data Size (in bytes)*
$00B0	*reserved for Apple use opcode[3,4]		0
⋮	⋮		
$00CF	*reserved for Apple use opcode		0
$00D0	*reserved for Apple use opcode + 4 bytes data length + data		4 + data length
⋮	⋮		
$00FE	*reserved for Apple use opcode + 4 bytes data length + data		4 + data length
$00FF	opEndPic	end of picture	2
$0100	*reserved for Apple use opcode[3,4] + 2 bytes data[7]		2
⋮	⋮		
$01FF	*reserved for Apple use opcode + 2 bytes data[7]		2
$0200	*reserved for Apple use opcode + 4 bytes data[7]		4
⋮	⋮		
$0BFF	*reserved for Apple use opcode + 4 bytes data[7]		22
$0C00	HeaderOp	opcode	24
$0C01	*reserved for Apple use opcode[3,4] + 4 bytes data[7]		24
⋮	⋮		
$7F00	*reserved for Apple use opcode + 254 bytes data[7]		254
⋮	⋮		
$7FFF	*reserved for Apple use opcode + 254 bytes data[7]		254
$8000	*reserved for Apple use opcode		0
⋮	⋮		
$80FF	*reserved for Apple use opcode		0
$8100	*reserved for Apple use opcode + 4 bytes data length + data		4 + data length
⋮	⋮		
$FFFF	*reserved for Apple use opcode + 4 bytes data length + data		4 + data length

1. The opcode value has been extended to a word for version 2 pictures. Remember, opcode size = 1 byte for version 1.
2. Because opcodes must be word-aligned in version 2 pictures, a byte of 0 (zero) data is added after odd-size data.
3. The size of reserved opcodes has been defined. They can occur only in version 2 pictures.
4. All unused opcodes are reserved for future Apple use and should not be used.
5. For opcodes $0040–$0044: rounded-corner rectangles use the setting of the OVSize point (refer to opcode $000B).
6. For opcodes $0090 and $0091: data is unpacked. These opcodes can only be used for rowBytes less than 8.
7. For opcodes $0100-$7FFF: the amount of data for opcode $nnXX = 2 * nn bytes.
8. See *Inside Macintosh* (1985).

Table C. Data format of version 2 PICT opcodes.

Opcode	*Name*	*Description*
$0012	BkPixPat	color background pattern
$0013	PnPxPat	color pen pattern
$0014	FillPixPat	color fill pattern

```
      IF patType = ditherPat
THEN
         patType: word;        { pattern type = 2 }
         pat1Data: Pattern;    { old pattern data }
         RGB:  RGBColor;       { desired RGB for pattern }
ELSE
         patType: word;        { pattern type = 1 }
         pat1Data: Pattern;    { old pattern data }
         pixMap:               { pixMap format shown below }
         colorTable:           { colorTable format shown below }
         pixData:              { pixData format shown below}
END;
```

$0015	PnLocHFrac	fractional pen position

PnLocHFrac: word:

If PnLocHFrac < > 1/2, it is always put to the picture before each text drawing operation.

$0016	ChExtra	extra for each character

ChExtra: word:

After ChExtra changes, it is put to picture before next text drawing operation.

$001A	RGBFgCol	RGB foreColor
$001B	RGBBkCol	RGB backColor
$001D	HiliteColor	RGB hilite color
$001F	OpColor	RGB OpColor for arithmetic modes

```
RGB:    RGBColor;    { desired RGB for foreground/background }
```

$001C	HiliteMode	hilite mode flag

No data. This opcode is sent before a drawing operation that uses the hilite mode.

$001E	DefHilite	use default hilite color

No data. Set hilite to default (from low memory).

The next four opcodes ($0090, $0091, $0098, $0099) are modifications of version 1 opcodes. The first word following the opcode is the rowBytes. If the high bit of the rowBytes is set, then it is a pixMap containing multiple bits per pixel; if it is not set, it is a bitMap containing one bit per pixel. In general, the difference between version 1 and version 2 formats is that the pixMap replaces the bitMap, a color table has been added, and pixData replaces the bitData.

Note: opcodes $0090 and $0091 are used only for rowBytes less than 8.

Table C (cont'd). Data format of version 2 PICT opcodes.

Opcode	*Name*		*Description*
$0090	BitsRect		copybits, rect clipped
	pixMap		{ described in Table D }
	colorTable:		{ described in Table D }
	srcRect:	rect;	{ source rectangle }
	dstRect:	rect;	{ destination rectangle }
	mode:	word;	{ transfer mode (may include new transfer modes) }
	pixData:		{ described in Table D }
$0091	BitsRgn		copybits, rgn clipped
	pixMap:		{ described in Table D }
	colorTable:		{ described in Table D }
	srcRect:	rect;	{ source rectangle }
	dstRect:	rect;	{ destination rectangle }
	mode:	word;	{ transfer mode (may include new transfer modes) }
	maskRgn:	rgn;	{ region for masking }
	pixData:		{ described in Table D }
$0098	PackBitsRect		packed copybits, rect clipped
	pixMap:		{ described in Table D }
	colorTable:		{ described in Table D }
	srcRect:	rect;	{ source rectangle }
	dstRect:	rect;	{ destination rectangle }
	mode:	word;	{ transfer mode (may include new transfer modes) }
	pixData:		{ described in Table D }
$0099	PackBitsRgn		packed copybits, rgn clipped
	pixMap:		{ described in Table D }
	colorTable:		{ described in Table D }
	srcRect:	rect;	{ source rectangle }
	dstRect:	rect;	{ destination rectangel }
	mode:	word;	{ transfer mode (may include new transfer modes) }
	maskRgn:	rgn;	{ region for masking }
	pixData:		{ described in Table D }

Table D. Data types found within new PICT opcode listed in Table C.

Opcode	*Name*		*Description*
pixMap =	baseAddr:	long;	{ unused = 0 }
	rowBytes:	word;	{ rowBytes w/high byte set }
	Bounds:	rect;	{ bounding rectangle }
	version:	word;	{ version number = 0 }
	packType:	word;	{ packing format = 0 }
	packSize:	long;	{ packed size = 0 }
	hRes:	fixed;	{ horizontal resolution (default = $0048.0000) }
	vRes:	fixed;	{ vertical resolution (default = $0048.0000) }
	pixelType:	word;	{ chunky format = 0 }
	pixelsize:	word;	{ no. of bits per pixel (1, 2, 4, 8) }
	cmpCount:	word;	{ no. of components in pixel = 1 }
	cmpSize:	word;	{ size of each component = pixelSize for chunky }
	planeBytes:	long;	{ offset to next plane = 0 }
	pmTable:	long;	{ color table = 0 }
	pmReserved:	long;	{ reserved = 0 }
	END;		
	colorTable = ctSeed:	long;	{ id number for color table = 0 }
	ctFlags:	word;	{ flags word = 0 }
	ctSize:	word;	{ number of ctTable entries –1 }
			{ ctSize + 1 color table entries }
			{ each entry = pixel value, red, green, }
			{ blue: word }
	END;		
	pixData: If rowBytes < 8 then data is upacked data size = rowBytes* (bounds. bottom-bounds. top) ; If rowBytes >= 8 then data is packed. Image contains (bounds. bottom-bounds. top) packed scanlines. Packed scanlines are produced by the packBits routine. Each scanline consists of [byteCount] [data]. If rowBytes > 250 then byteCount is a word, else it is a byte.		
	END;		